

C++11 - określanie typów

Piotr Beling

Uniwersytet Łódzki, Wydział Matematyki i Informatyki

30 kwietnia 2013

Streszczenie

Niniejszy artykuł jest jednym z serii artykułów w których zawarto przegląd nowych elementów języka C++ wprowadzonych przez standard ISO/IEC 14882:2011, znany pod nazwą C++11.

W artykule przedstawiono nowe możliwości związane z określaniem typów zmiennych. Opisano słowa kluczowe **auto** i **decltype**, nową składnię deklarowania funkcji/metod oraz narzędzia zawarte w pliku nagłówkowym `<type_traits>`.

Spis treści

1	Wstęp	2
2	AUTOmatyczna dedukcja typu	2
3	Obliczanie typu wyrażenia za pomocą <code>decltype</code>	3
4	Nowa składnia deklarowania funkcji/metody	3
5	Nagłówek <code><type_traits></code>	4

1 Wstęp

Standard języka C++, znany jako C++11 (a także C++0x), został opublikowany we wrześniu 2011 roku, w dokumencie ISO/IEC 14882:2011¹.

W stosunku do poprzedniego standardu (C++03), wprowadza on dużą liczbą nowych elementów, zarówno do samego języka, jak i do jego standardowych bibliotek.

W większość popularnych kompilatorów dość szybko zaimplementowano prawie wszystkie nowe elementy wprowadzone w C++11.

Niniejszy artykuł, jeden z serii, stanowi krótki i bogato zilustrowany przykładami przegląd nowych konstrukcji które wzbogaciły język.

Szczególną uwagę przywiązano w nim do możliwości związanych ze wskazywaniem typów zmiennych.

2 AUTOMatyczna dedukcja typu

Słowo kluczowe **auto**, mało istotne w C++03, ma w C++11 nowe znaczenie i umożliwia m.in. dedukcję typu zmiennej na podstawie typu wartości inicjującej:

```
auto i = 5;  
const auto c = 8;
```

oznacza to samo co:

```
int i = 5;  
const int c = 8;
```

gdź 5 oraz 8 są typu **int**.

Następujący kod (gdzie `v` jest typu `std::vector<int>`):

```
for (std::vector<int>::iterator i = v.begin();  
      i != v.end();  
      ++i)  
    //...
```

można w C++11 zapisać krócej:

```
for (auto i = v.begin(); i != v.end(); ++i)  
    //...
```

Jeśli po prawej stronie operatora przypisania pojawi się referencja, **auto** będzie domyślnie oznaczało typ pozbawiony tej referencji. Referencję, jak i inne modyfikatory, można jednak dopisać:

```
int& f();  
auto i = f();           // i jest typu int  
auto& r = f();          // r jest typu int&  
const auto c = f();     // c jest typu const int
```

¹Dokument ten dostępny jest odpłatnie, jednakże ze strony <http://www.open-std.org> można pobrać jego darmową wersję roboczą [5]. Najnowsza (N3337) datowana jest na 16 stycznia 2012.

3 Obliczanie typu wyrażenia za pomocą `decltype`

Dzięki operatorowi `decltype` możliwe jest użycie typu podanego wyrażenia, np.

```
//int i;  
decltype(i) j = i; //j jest typu int  
std::vector<decltype(i)> v(3, i); //wektor z 3 kopiami i  
decltype(v)::iterator iter; //odpowiedni dla v iterator
```

Samo wyrażenie wewnątrz `decltype` nie jest wartościowane i w związku z tym nie powoduje żadnych efektów ubocznych, np.:

```
int i = 1;  
decltype(i++) j; // j jest typu int  
assert(i == 1);
```

Zachowanie `decltype` zostało tak określone, by pokrywało się z intuicją większości programistów, jednocześnie dając jak największe możliwości. Dokładana definicja typu `decltype(e)` jest jednak dość skomplikowana [4]:

- jeśli wyrażenie `e` jest odwołaniem do zmiennej w zakresie lokalnym lub w przestrzeni nazw, statycznej zmiennej w klasie lub parametru funkcji, to wynikiem jest zadeklarowany typ zmiennej lub parametru, np.

```
int i;  
struct A { double x; };  
const A* a = new A();  
decltype(i) x1; // typ to int  
decltype(a->x) x2; // typ to double
```

- jeśli `e` jest wywołaniem funkcji lub użyciem przeciążonego operatora, `decltype(e)` oznacza zadeklarowany typ wyniku tej funkcji, kontynuując przykład:

```
const int&& foo();  
decltype(foo()) x3; // typ to const int&&
```

- w przeciwnym razie, jeśli `e` to l-wartość, `decltype(e)` to `T&`, gdzie `T` jest typem `e`; jeśli `e` jest r-wartością, to wynikiem jest `T`, kontynuując przykład:

```
decltype((a->x)) x5; // typ to const double&
```

Proszę zwrócić uwagę na dodatkowe nawiasy które pojawiły się w deklaracji `x5` w stosunku do `x2`. Wymuszają one potraktowanie wyrażenia jako l-wartość.

4 Nowa składnia deklarowania funkcji/metody

W C++11 następujące deklaracje są tożsame:

```
int f(int x, double y) { ... } //C++03 i C++11
auto f(int x, double y) -> int { ... } //tylko C++11
```

Nowa składnia deklarowania funkcji/metody została wprowadzona do C++11 by umożliwić wygodne wyliczenie typu zwracanego przez funkcję/metodę na podstawie typów jej argumentów, np.:

```
//Niech vec<T> będzie klasą reprezentującym wektor,
//mającą dwa pola typu T o nazwach x, y.
//Przykładowa implementacja operatora dodawania wektorów,
//z odpowiednim awansem typów:
template <typename A, typename B>
auto operator+(vec<A> a, vec<B> b) -> vec<decltype(a.x+b.x)> {
    return vec<decltype(a.x+b.x)>(a.x+b.x, a.y+b.y);
};
//... przykład użycia:
vec<double> a(1.5, 2.7);
vec<int> b(1, 3);
auto c = a + b; //c == vec<double>(2.5, 5.7);
```

Proszę zauważyć, że następujący zapis nie jest prawidłowy gdyż wewnątrz **decltype** argumenty a i b nie są jeszcze zdefiniowane:

```
vec<decltype(a.x+b.x)> operator+(vec<A> a, vec<B> b) //...
```

Prawidłowy, choć mniej wygodny od pierwowzoru, jest za to zapis:

```
#include <utility> //dla std::declval
//...
template <typename A, typename B>
vec<decltype(std::declval<A>()+std::declval<B>())>
operator+(vec<A> a, vec<B> b) {
    return vec<decltype(a.x+b.x)>(a.x+b.x, a.y+b.y);
};
```

Zdefiniowany w nagłówku `utility` szablon `std::declval` zwraca r-wartość typu podanego jako parametr szablonu, co umożliwia np. wskazanie pól tego typu. Typ ten może być niekompletny lub abstrakcyjny, zaś jego obiekty niemożliwe do utworzenia. Jednakże `std::declval` można użyć jedynie w operatorach które nie wartościują wyrażeń, takich jak **decltype** czy **sizeof**.

5 Nagłówek <type_traits>

Nagłówek <type_traits> zawiera zestaw narzędzi, głównie szablonów klas, pozwalających sprawdzać własności typów oraz tworzyć nowe typy poprzez modyfikacje istniejących (np. usunięcie referencji czy modyfikatora **const**).

Większość szablonów klas zdefiniowanych w <type_traits> specjalizowana jest jednym typem.

Szereg szablonów klas, których nazwy zaczynają się od `is_` (rzadziej od `has_`) posiada jedno statyczne pole typu **bool** o nazwie `value` które ustawiane jest na **true** tylko gdy typ będący parametrem ma daną własność. Na przykład:

```
| std::is_abstract<T>::value // T jest klasą abstrakcyjną?
```

Są one szczególnie użyteczne z `std::enable_if` do wybrania implementacji funkcji lub metody w zależności od własności typu. Na przykład niżej zdefiniowany szablon funkcji kopiuje tablicę używając efektywnego `memcpy` gdy tablica jest typów które można bezpiecznie skopiować tą metodą (mają trywialny operator kopiowania):

```
| template<typename T>
| typename std::enable_if<
|     std::is_trivially_copy_assignable<T>::value
| >::type
| mycopy(const T* source, T* dest, std::size_t count) {
|     memcpy(dest, source, count*sizeof(T));
| }
|
| template<typename T>
| typename std::enable_if<
|     !std::is_trivially_copy_assignable<T>::value
| >::type
| mycopy(const T* source, T* dest, std::size_t count) {
|     for(std::size_t i = 0; i < count; ++i)
|         *dest++ = *source++;
| }
```

Gdy pierwszym argumentem szablonu `std::enable_if` jest **true**, to zawiera on pole `type` określające typ drugiego argumentu (domyślnie **void**), w przeciwnym razie jest on pustą klasą. W ten sposób, dla dowolnego typu, tylko jedna z wersji `mycopy` będzie poprawnie zdefiniowana i użyta (istnienie drugiej nie spowoduje błędu kompilacji²).

Szablony klas których nazwy, typowo, zaczynają się od `add_`, `remove_` oraz `make_` udostępniają, w polu `type`, zmodyfikowaną wersję typu będącego argumentem. Na przykład:

```
| typename remove_const<T>::type
| // np. int gdy T jest const int
```

Szablon `std::result_of` służy do ustalenia wartości zwracanej przez podany wskazany funktor:

```
| typename std::result_of<fun(int)>::type
| // typ zwrócony przez fun dla argumentu typu int
```

²Ta zasada znana jest pod nazwą SFINAE (Substitution failure is not an error).

Literatura

- [1] C++ reference. <http://en.cppreference.com>. [dostęp: 2013-04-27].
- [2] Wikipedia, the free encyclopedia: C++11. <http://en.wikipedia.org/wiki/C++11>. [dostęp: 2013-04-27].
- [3] Wikipedia, wolna encyklopedia: C++11. <http://pl.wikipedia.org/wiki/C++11>. [dostęp: 2013-04-27].
- [4] Wikipedia, wolna encyklopedia: decltype. <http://pl.wikipedia.org/wiki/decltype>. [dostęp: 2013-04-27]. 3
- [5] ISO. *ISO/IEC N3337=12-0027 Working Draft, Standard for Programming Language C++*. International Organization for Standardization, 2012. 1